Auto-parallelization of Pidgin C

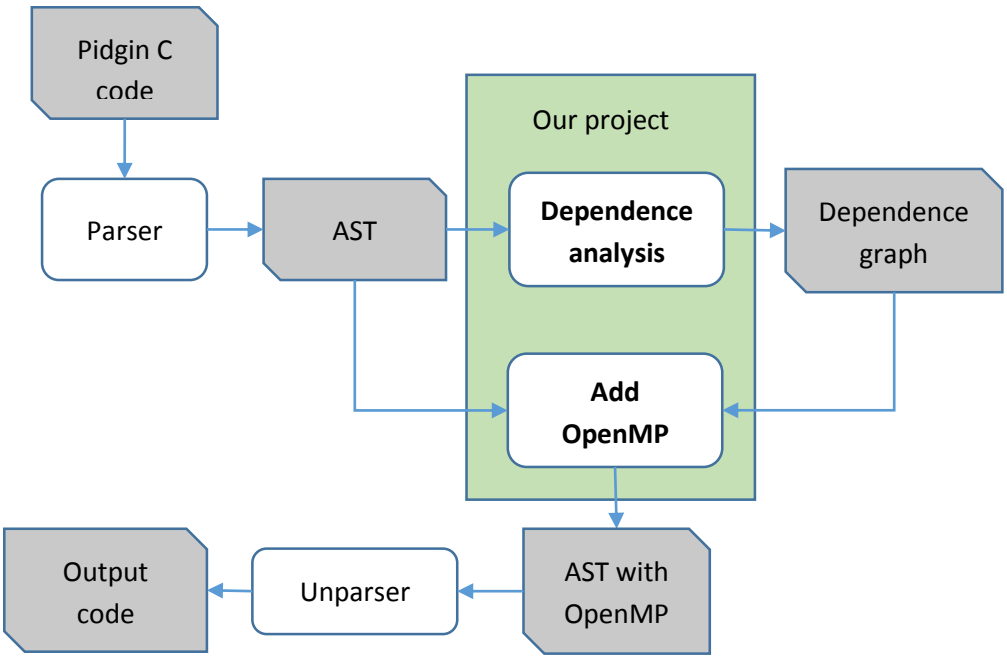
- **P523 final project**

Tanghong Qiu (taqiu@indiana.edu) **Zi Wang** (wang417@indiana.edu)

1. Introduction and Background

Parallelization is an importance approach to utilize multiple processors and reduce the running time of program. The goal of this project is to implement an auto-parallelization complier for Pidgin C. This project will involve detecting Pidgin C loops that can be parallelized and generating OpenMP directives that allow those loops to be run in parallel. OpenMP is a set of compiler directives that help the application developer to parallelize their workload, and this project mainly utilize the "for directive" which helps share iterations of a loop between a group of threads.

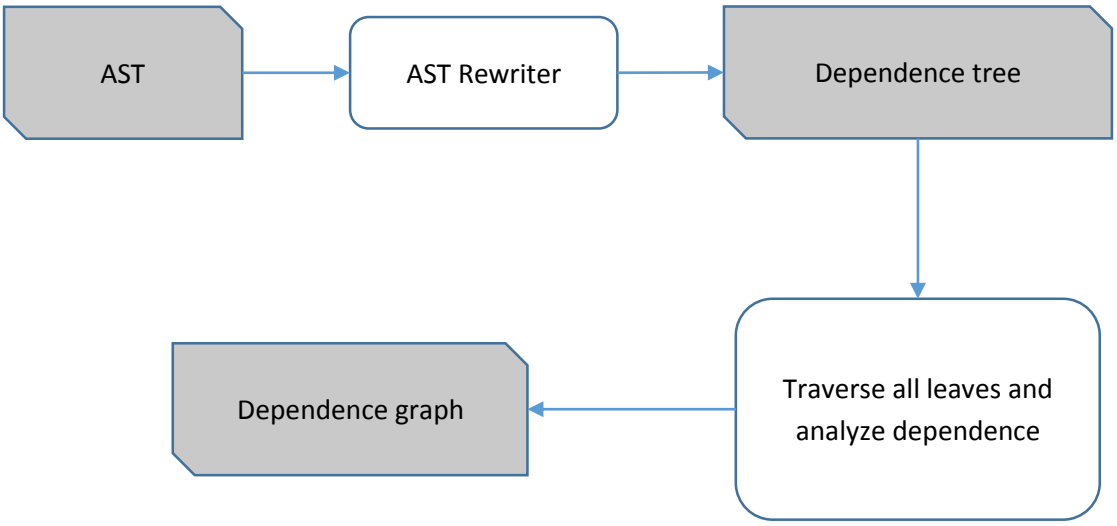
According to the loop parallelization theorem, it is valid to convert a sequential loop to a parallel loop if the loop carries no dependence [1]. Therefore, to detect the parallelizable loop, the compile need to analyze the dependence of target loop, which is the major work of this project. This step will generate a dependence graph which is used to figure out the outermost parallelizable loop and add OpenMP directives.



The project is based on our previous parser assignment, and this project will analyze the abstract syntax tree from Pidgin C parser. This project output a new AST with OpenMP

directive according to the dependence graph, and unparse the AST to Pidgin C finally.

2. Dependence analysis



2.1 AST Transformation

The abstract syntax tree is the original intermediate representation for Pidgin C. The dependence graph is able to be derived from the AST directly, but the AST contains a lot of useless information for dependence analysis. Actually, the dependence analysis is concerned only with the variables in expression or statement.

Firstly, all statements should be simplify to a uniform intermediate representation which only keep the variable and its subscript, and we define several class to save those information. The id field of statement class is a unique label for all statement, and the left and right field will save the variables on the assignment left hand side and right respectively. The variable class will save the identifier and subscripts of variables.

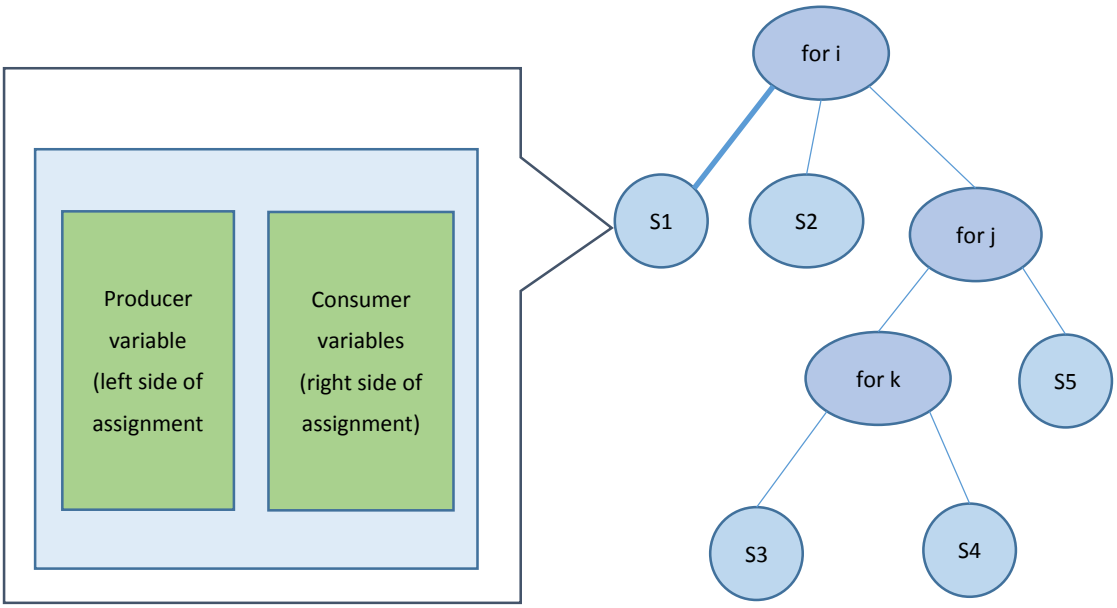
```
class Statement
    def initialize(id, left, right)
        @id = id
        @left = left
        @right = right
    end
```

```
class Variable
    def initialize(id, subs)
        @id = id
        @subs = subs
    end
end
```

```
class Subscript
    def initialize(var, offset)
        @var = var
        @offset = offset
    end
end
```

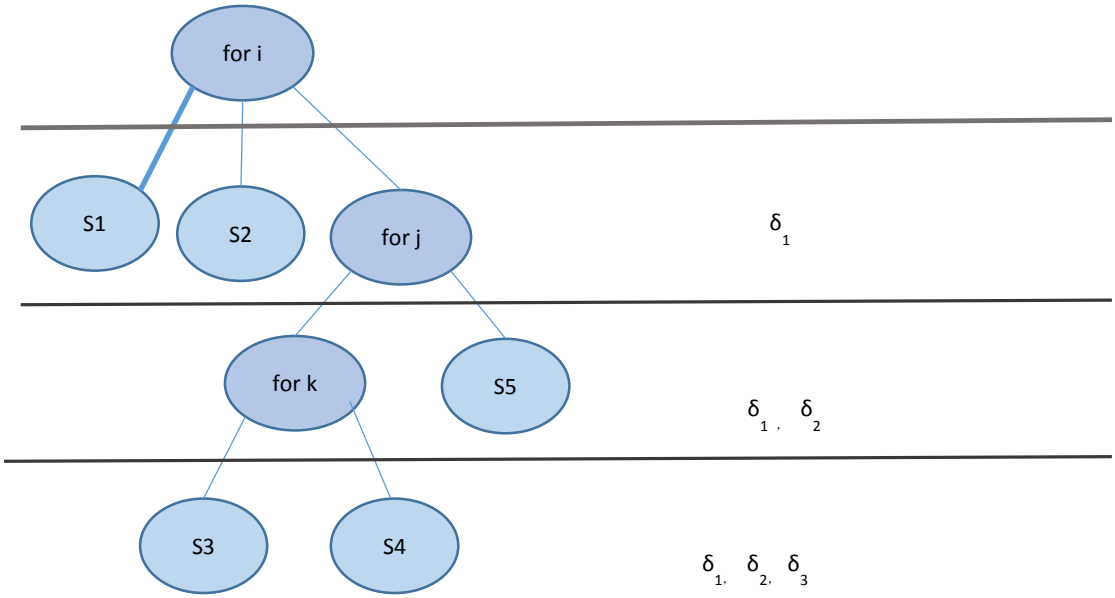
Beside simple statements, the compound statements (e.g. if…else.. , while…) are also allowed in the "for loop". The statements in the block are addressed as simple statements, and the conditions of compound statements are regarded as statements which only have right side variables.

The statements are stored in a tree. The root node is a "for loop" node which contain a list of statement at its level or nested for loop nodes. Initially, we use a link list to represent the relationship of statement, but the link list cannot represent complicate "for loop" structure. Therefore, we use a tree which can exactly represent the sequence and level of statement.



2.2 Tree Traversal Algorithm

After generating the tree for dependence analysis, we need to develop a tree traversal algorithm to take out every two node and analyze their dependence. First of all, we must guarantee the entire tree are traversed and no node is left out. Secondly, we have to reduce the duplicate analysis.



From the above figure, we can see that the different level of loops exits different level dependences. The loop-carried dependence of statements only exist at current or above level. Therefore, we employ the depth first algorithm to traverse every node the tree. Meanwhile, we generate a list of statement nodes at current and deep level, and analyze the dependence between the node and the list. To minimize the time of analysis, the list only contains the nodes with equal or greater id and the nodes at deep level.

S1	<->	S1, S2, S3, S4, S5	dependence at level [i]
S2	<->	S2, S3, S4, S5	dependence at level [i]
S3	<->	S3, S4	dependence at level [i, j, k]
S4	<->	S4	dependence at level [i, j, k]
S5	<->	S3, S4, S5	dependence at level [i ,j]

2.3 Dependence Analysis Algorithm of two statements

- The tree iteration mechanism can make sure all the statements can be involved into consideration of parallelization. In fact when we iterate the tree, we choose two elements (statements) out from the tree and analyze the dependence in it. In dependence analysis, our task is:
- 1, determine whether the two statements are independent with each other
- 2, if the two statements are not independent, locate which level the dependence lie.
- So we need an algorithm to detect dependences hidden in two statements and tell the dependence level. In this section we would introduce our implement of the algorithm.
- There are two kinds of dependences that may occur in a “for” loop. First is the inner-loop dependence. Since the minimum granularity of our parallelization is each

iteration within the same iteration space, we would not change the sequence of statements in the same iteration. We would regard inner-loop dependence no harm to parallelization. The second is the loop-carry dependence. Each statement in a "for" loop will execute $\prod_{i=1}^{n} N_i$ times, where Ni is the total iteration time of level i. However, the semantics and logic may also embed in the sequence of the same statement. Our parallelization, however, would disorder the iteration-sequence of such identical statement. So loop-carry dependence is within our consideration.

- In fact, our dependence analysis algorithm mainly focuses on detecting the level of loop-carry dependence. In C language, there are two kinds of semantic circumstances that would raise loop-carry dependence. One is dependence in array variable that contains varied index. For example, array variables A[i][j] and A[i][j+1] have common memory writing in consecutive iteration times. In this circumstance, we can calculate directive vector to find and locate the dependence. Another occasion is dependence lying in a variable that has a constant memory address throughout the iteration. Let's look at the following code:
- for i=1:N
- c = B[i]; //S1
- end
- We cannot detect dependence by directive vector analysis on S1, because there is no varied index for us to get directive vector. In more interesting case like following:
- for i=1:N
- c[0][0]= B[i]; //S1
- end
- If we use directive vector, we will find DV(S1, S1) = (=, =). It seems no dependence there. But in fact we have output dependence in S1. The value of c[0][0] is dependent on the B[i] of last executing iteration. When we parallelize the "for" loop, the last execution of B[i] will be not sure. Race condition will appear in this case. In order to detect this kind of output dependence, we need to use a method called race-condition detector.
- We create function dependence(stmt1, stmt2, index_list) to implement the "atom-operation" of dependence analysis. We input two statements stmt1 and stmt2 and tell the function which loops does stmt1 and stmt2 lay in. We use loop index name to represent loop. For example index_list = {i, j, k} means there are loop i, j, k that cover the two statement. The function get to know that there is at most three level of dependence among these two statements. If stmt1 and stmt2 are not in the same level of loop, we would only push into index_list the index that the lower level statement is involved. For example, stmt1 is in level 2, but stmt2 is in level 3, we would push i, j into index_list (no k), because we know stmt2 doesn't have index k, so it is impossible for them to have level k independence. dependence(stmt1, stmt2, index_list) return a set of dependence graph, which have been introduced in above section. We use a set called "edges" to restore the total dependence relation found in two statements.
- In function dependence(stmt1, stmt2, index_list), we have two separate function model to be implemented. One is directive vector analysis, another is race-condition detector.
-

2.4 The implement of directive vector analysis

- Given a statement S, the variables in it can be divided into left value and right value. Usually in each statement we can get one left value and a set of right values. If we need to use directive vector analysis between two statements S1 and S2, we need to pick left value of each S1 and S2 and compare it to
- 1, all the right values of another statement
- 2, left value of another statement
- In each of the comparison process, we first check whether the array names of two variables are the same. If they are not the same, that means the two variables are impossible to store data in the same memory address. There is no dependence on these two variables. However if the variable names are the same, we need to get directive vector between these two variables and get to know which level carries dependence. We create function getDepLevel(var1, var2, index_list) to finish this task. var1 and var2 are two variables that is being compared. It returns a set of integer numbers that represent the levels of dependence. If there is no dependence between two variable, depLevel would be empty. Here is the implement of function getDepLevel().

```
190     def getDepLevel(var1, var2, index_list)
191         depLevel=[]
192         get=false
193         if var1.id.eql? var2.id
194         index_list.length.times{|i|
195             if get == true
196                 break;
197             end
198             index = index_list[i]
199             var1.subs.length.times{|j|
200                 if var1.subs[j].var.eql? index
201                     if var2.subs.length>j
202                         if var2.subs[j].var.eql? var1.subs[j].var
203                             if var2.subs[j].offset - var1.subs[j].offset != 0
204                                 depLevel.push i+1
205                                 get=true
206                             end
207                         else
208                             depLevel.push i+1
209                             get=true
210                         end
211                     end
212                 end
```

-

-

```
213
214             }
215
216             var2.subs.length.times{|j|
217                 if var2.subs[j].var.eql? index
218                     if var1.subs.length>j
219                         if var1.subs[j].var.eql? var2.subs[j].var
220                             if var1.subs[j].offset - var2.subs[j].offset != 0
221                                 depLevel.push i+1
222                                 get=true
223                             end
224                         else
225                             depLevel.push i+1
226                             get=true
227                         end
228                     end
229                 end
230
231             }
232         }
233         end
234
235         depLevel= depLevel.uniq
236     end
```

- The principle of the function is to locate the level of index by its sequence in index_list. Then we do an index variable and index offset deduction of the same dimension of two variables to get the directive component of two variables. Then we connect the level information and directive component together to form the directive vector. From the vector we choose the first "<" level to be dependent.
- The line 193 checks whether var1 and var2 have the same names. If their names are the same, we keep the process, if the names are different, we would stop checking. Then we iterate all the for-loop indexes in index_list (line 194) and search whether each index i will appear in one variable (both var1 and var2, line 119~120, 216~217). If we find the specific index variable in a particular dimension of the array (line 200), for example dimension j, we would get the index condition (index variable and index offset) of the same dimension of the other variable and get the directive component value (line 202 and 203). If the value is "<", we will be sure a dependence lay on index i. Then according to the location of index i in index_list, we fix the level of dependence (204). After we get the dependence level, we would not keep calculating the rest of directive component (line 195~197).
- With the function getDepLevel(), it is very easy for dependence() to make dependence analysis. For example the way to do left value – left value analysis, we can do this:

```
84          ####### left value of both stmts
85          if stmts1.left!=nil && stmts2.left!=nil
86              var1=stmts1.left
87              var2=stmts2.left
88              depLevels = depLevels+(getDepLevel var1, var2, index_list)
89          end
90
```

- We can see all we need to do is to call this function and we can get the dependence information of output dependence. However, there is some case that the statements don't have left value, for example "if(x<5)", and others like such control statements. Then their left value is nil. We would not calculate dependence of those which don't have left value

(line 85).

```
92          ######## left value of s1 and right values of s2
93          if stmts1.left!=nil
94              var1=stmts1.left
95              vars=stmts2.right
96              vars.each{|var2|
97                  depLevels = depLevels+(getDepLevel var1, var2, index_list)
98
99              }
100         end
```

- When we calculate dependence on one left-value and the other statement's right values, we need to iterate all variables in right values set (line 96) and call function getDepLevel().

```
111         depLevels = depLevels.uniq
112                 depLevels.each{|i|
113             if not existEdge? edges, stmts1.id, stmts2.id, i
114                 edges.push Edge.new stmts1.id, stmts2.id, i
115             end
116
117         }
```

- As a result we would get a set "depLevels" that contains duplicated dependence level. First we delete the duplicated elements of depLevels (line 111). To each element of it, we create edge information of dependence based on the statements name and dependence level unless there is no such edge existing.
-

2．5 Race condition detector

- Sometimes we would meet such an occasion: we have statement S in loop level {i, j}, but index j doesn't appears in the left value of S, but it appears in the right value of S. In this case if the variable names are not same between left value and right value, we cannot detect any dependence by merely directive vector analysis. For example: a[i]= b[i-1][j+1]+d[i]; //S. directive vector analysis is inclined to let S parallelizable at any level, but in fact a dependence lays in level j. Because in the view of level j a[i] is a constant memory address, however, b[i-1][j+1] is an array. When we iterate an array and write the value to a constant memory address, we would get loop-carried output dependence. When we force to parallelize it, we will get race condition. In order to detect such a kind of potential race condition, we create Race Condition Detector (RCD).
- We implement RCD mainly by function checkRaceCondition(stmt, index_list). Parameter "stmt" is the Statement object passed into. index_list comes from index_list in function dependence(). The code of checkRaceCondition() is as following:

```
144     def checkRaceCondition(stmt, index_list)
145         depLevels=[]
146         index_list.length.times{|i|
147             find=false
148             if not checkIndex stmt.left, index_list[i]
149                 find=true
150                 stmt.right.each{|var|
151                     if checkIndex var, index_list[i]
152                         depLevels.push i+1
153                         find=false
154                     end
155                 }
156             end
157             if find==true
158                 if checkIdfdVar(stmt)
159                     depLevels.push i+1
160                 end
161             end
162         }
163         depLevels
164
165     end
```

- The principle of the function is to make sure all the indexes that appear in the right values must also appears in the left value. If it fails to obey this rule, the function would regard a dependence embedded in this level. Line 146 iterate all the index i among index_list. Then we find out all indexes that fail to appear in the left value (line 148). For example index i is in such occasion, the function would detect whether the index appears in one of the right values (line 150). If it indeed does, we would put this level into depLevels, which carries the set of levels that has dependences (line 152).
- However there is an exception of such dependence. When there is a non-array variable on both left value and right values, race condition also may happen when parallelizing. In order to solve this problem we add an indicator named “find” to find such a kind of variable. When find is true, it means both left value and right values contains a non-array variable. Furthermore we would check if these two variables are in the same name (line 158~160). If their names are the same we would add the level i into depLevels.

```
119         depLevels=[]
120         if stmts1.left!=nil
121             depLevels = checkRaceCondition(stmts1, index_list)
122                     depLevels = depLevels.uniq
123                     depLevels.each{|i|
124                 if not existEdge? edges, stmts1.id, stmts1.id, i
125                     edges.push Edge.new stmts1.id, stmts1.id, i
126                 end
127
128             }
```

- As we can see, we just need to call checkRaceConditon() to get the levels of dependence (line 121) and then same as the process of directive vector analysis above, function dependence() update edges from the result of RCD.

2.4 Dependence Graph

A general representation for graph is using vertex set and edge set. (G = <V, E>) In this project, we don't care about the nodes which have no dependence with other nodes, so we only need to keep the edge set.

Edge = {statement1, statement2, level}

In the project, the types of dependences are not important for adding Open directive either. However, the level of dependence is required, so we need to save it.

3. Adding OpenMP directive

After generating the dependence graph, our compile will generate a new AST with OpenMP directive according to the original AST and the dependence graph. The strategy is to find the outermost level loop which have no dependence at this level. And then we will use the Pidgin C unparser to transform the AST back to Pidgin C code.

First, a hash table is used to count the number of dependences at each levels. Secondly, the compile traverses the original AST tree from the outer level to the inner level. If no dependence exists at a "for loop" level, an OpenMP for directive is added to this level. The remained child node won't be traversed if a directive has been added.

4. Conclusion and further work

All in all, the compile works well with most examples in the text book. However, the compiler only support the general type subscript (e.g. [i-2], [j], [k+3]), and subscripts like [i+k] or [9-j] are not supported in our project.

If a "for loop" node have more than one sub "for loops" at the same level, the compiler can't generate correct dependence graph for this case. Because we don't have a feasible approach to represent the dependence graph between the parallelized sub "for loops". The dependences between the parallelized sub "for loops" are more difficulty to decide, because the indexes of loop can be used repeatedly.

The adding OpenMP directive process can also be improved further. Initially, we decided to develop an algorithm from the codegen algorithm, but we realized the vectorization is different from the parallelization. The parallelization requires no dependence at the parallelized level. One of the removing dependence approach is to separate the loop, but this method will make the program create and join the threads repeatedly. It might reduce the

performance of program. Another advanced transformation is to change the position of for loop header, for example, the header of no-dependence for loop can be moved to outermost level and be parallelized. This approach can improve the performance of parallelization.

As we discussed above, this project can be extended further in different aspects. But the time for project is limited, so we just use the most reliable approach to do the auto-parallelization.

Reference

[1] Allen & Kennedy, Optimizing Compilers for Modern Architectures, 1st Edition